MODULE-2

STRUCTURAL TESTING

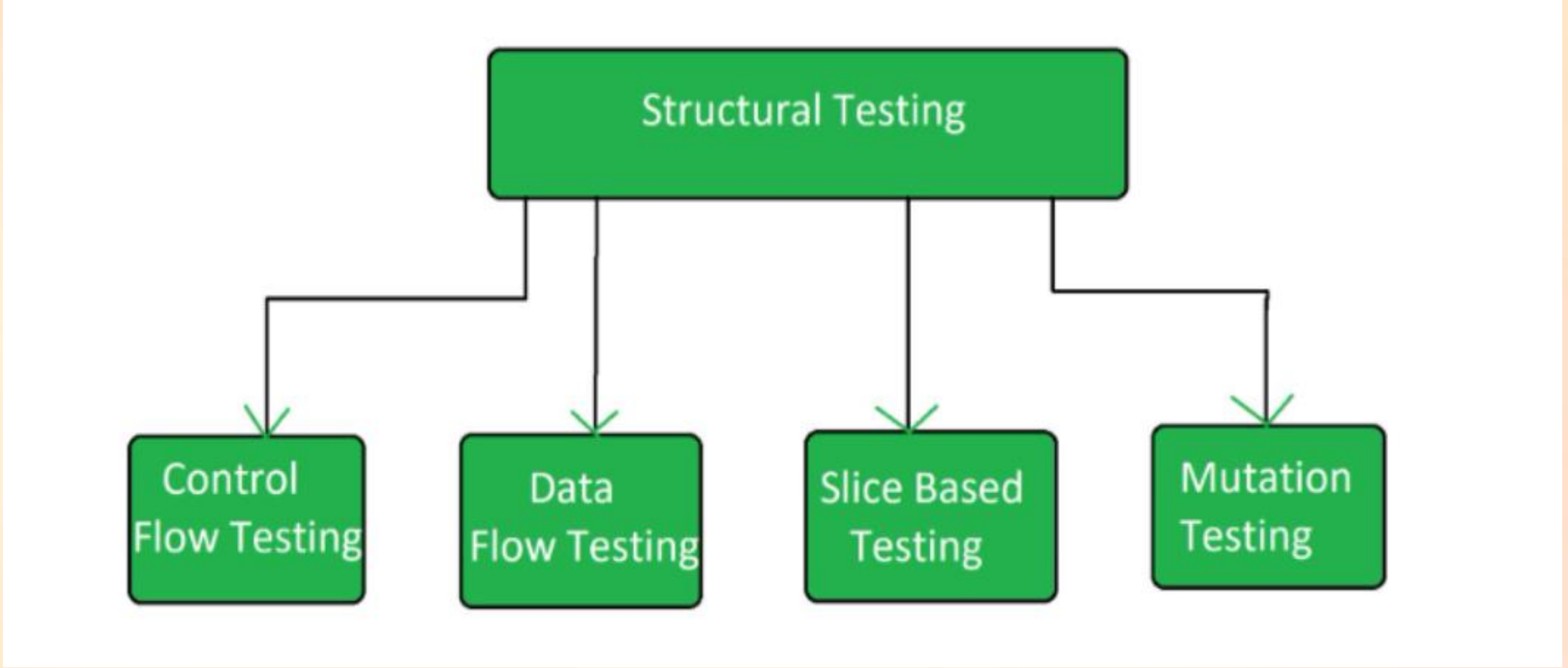
White-box Testing:

- The primary goal of white box testing is to focus on the flow of inputs and outputs through the software and to strengthen the security of the software.
- The white box testing contains various tests, which are as follows:
- ● Path testing
- ● Loop testing
- ● Condition testing
- ● Testing based on the memory perspective
- ● Test performance of the program

Structural Testing:

- The structure-based testing technique is also known as the 'white-box' or 'glass-box' testing technique because the testers require knowledge of how the software is implemented and works.
- Structural testing is basically related to the internal design and implementation of the software i.e. it involves the development team members in the testing team.
- It basically tests different aspects of the software according to its types.

Types of Structural testing

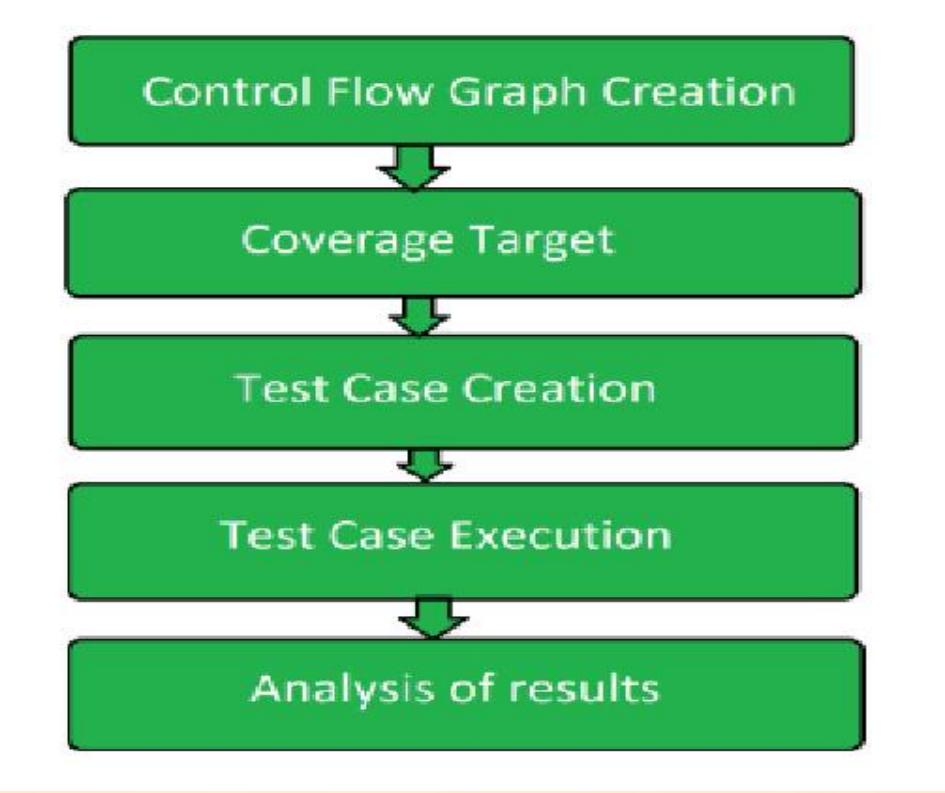


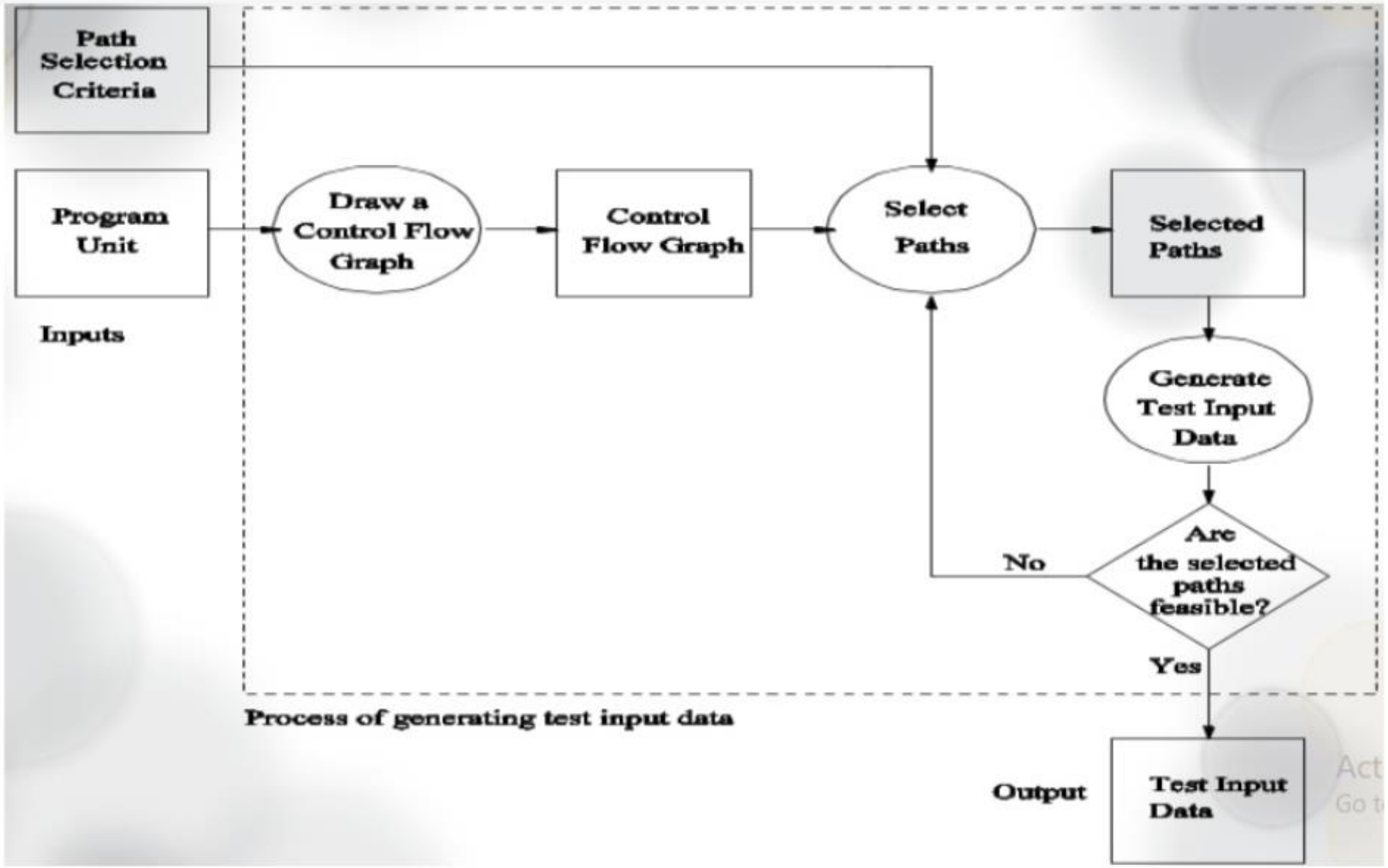
- **Control flow testing :** is a type of structural testing that uses the program's control flow as a model.
- The entire code, design, and structure of the software have to be known for this type of testing.
- Often this type of testing is used by the developers to test their own code and implementation.
- This method is used to test the logic of the code so that the required result can be obtained.
- **Data Flow Testing :** It uses the control flow graph to explore the unreasonable things that can happen to data.
- The detection of data flow anomalies is based on the associations between values and variables.
- Without being initialized usage of variables. Initialized variables are not used once.

- **Slice-Based Testing:** It was initially proposed by Weiser and Gallagher for software maintenance.
- It is useful for software debugging, software maintenance, program understanding, and quantification of functional cohesion.
- It divides the program into different slices and tests that slice which can majorly affect the entire software.
- **Mutation Testing:** Mutation Testing is a type of Software Testing that is performed to design new software tests and also evaluate the quality of already existing software tests.
- Mutation testing is related to modifying a program in small ways. It focuses to help the tester develop effective tests or locate weaknesses in the test data used for the program.

Control Flow Testing

- Control Flow Testing Control flow testing is a structural testing strategy that uses a program's control flow as a model.
- Following are the steps involved in the process of control flow testing:




Path Selection Criteria
Program Unit
Inputs
Draw a Control Flow Graph
Control Flow Graph
Select Paths
Selected Paths
Generate Test Input Data
Are the selected paths feasible?
No
Yes
Process of generating test input data
Output
Test Input Data

- **Node:** It represents one or more procedural statements.
- **Edges or Links:** They represent the flow of control in a program.
- **Decision node:** A node with more than one arrow leaving is called a decision node.
- **Junction Node:** A node with more than one arrow entering it is called a junction.
- **Regions:** Areas bounded by edges and nodes are called regions.

- The Cyclomatic complexity can be measured by knowing the number of independent paths ina program's source code, the formula is:
- $V(G)=E-N+2P$

OR

- $V(G)= D+1$

OR

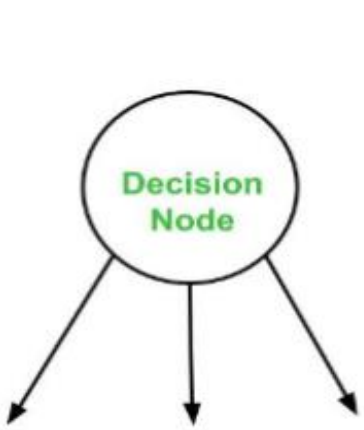
- $V(G)=R+1$

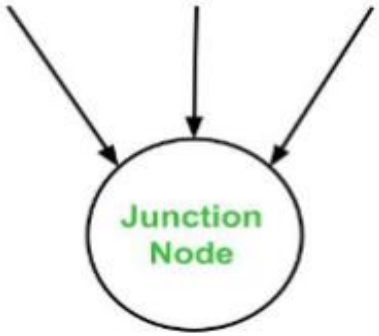
Where,

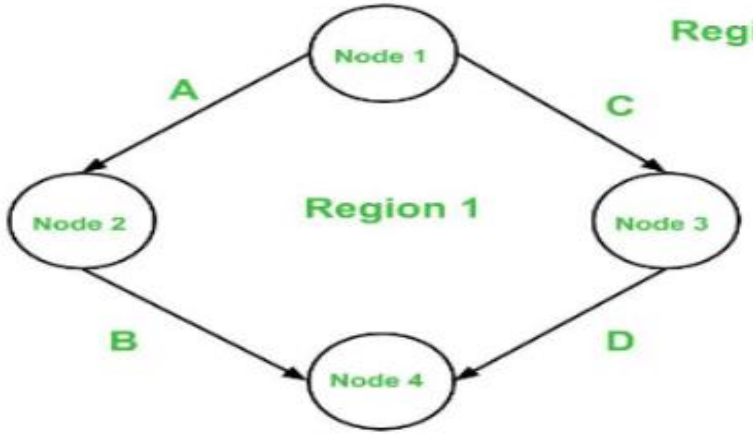
E=edges, N=nodes, P= components (no. of exit points),D=Decision node,

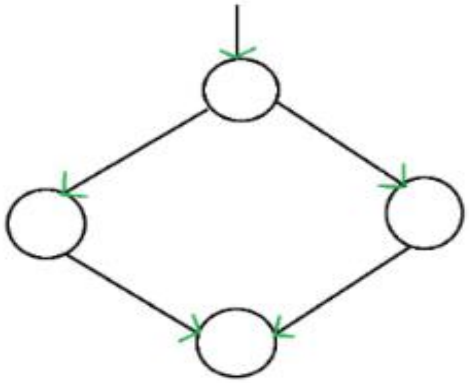
R= enclosed regions.

Basics on control graph creation

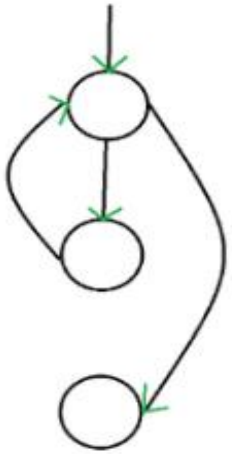




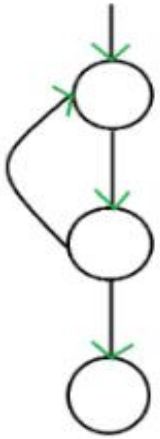




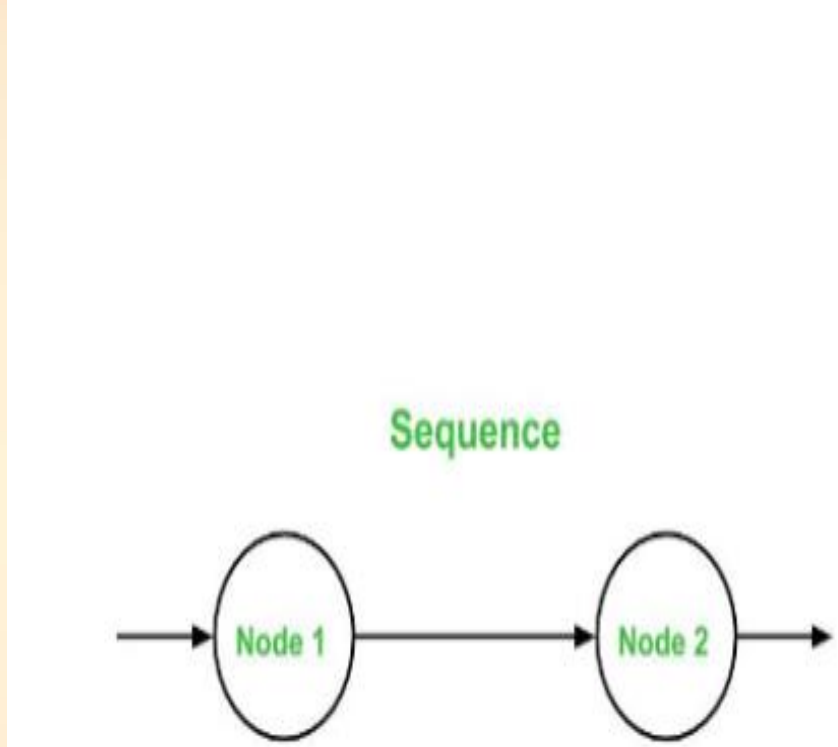
If-then-else

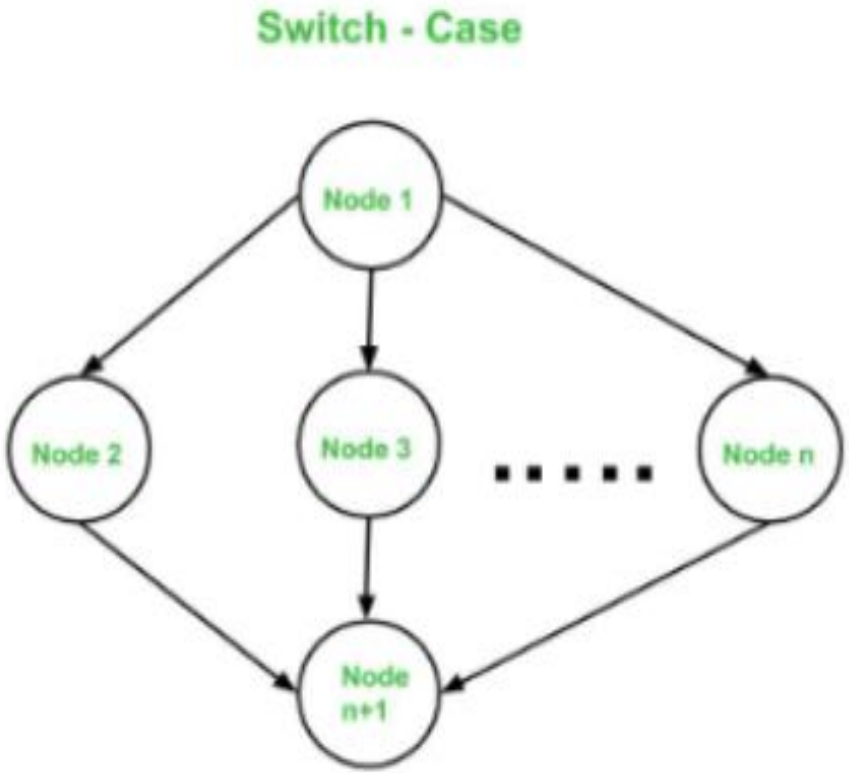


while



do-while


Sequence
Node 1
Node 2

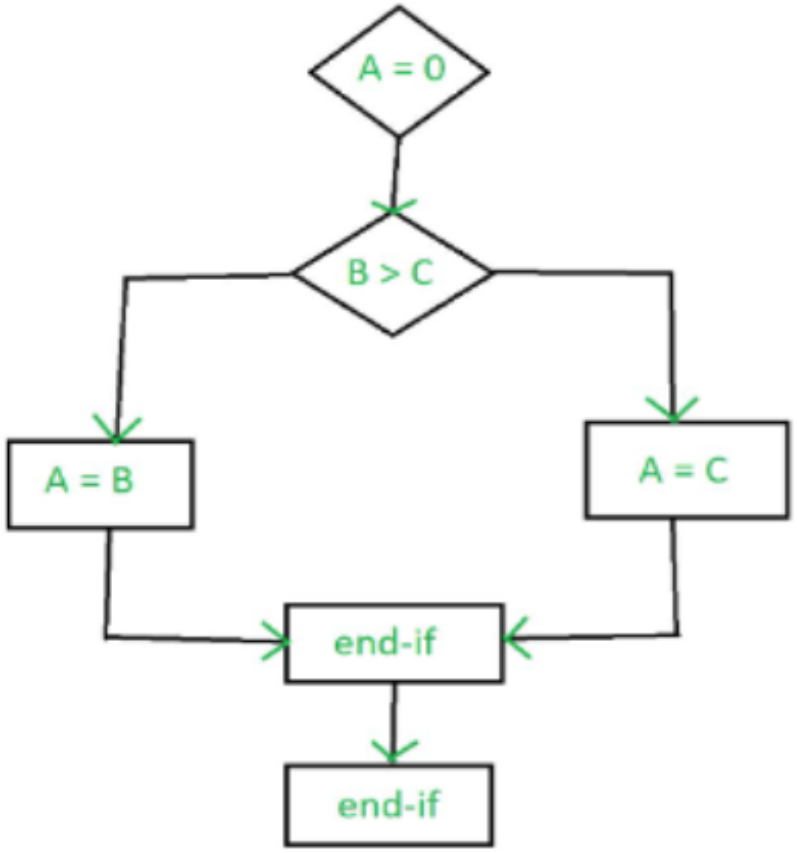

Switch - Case
Node 1
Node 2
Node 3
Node n
Node
n+1

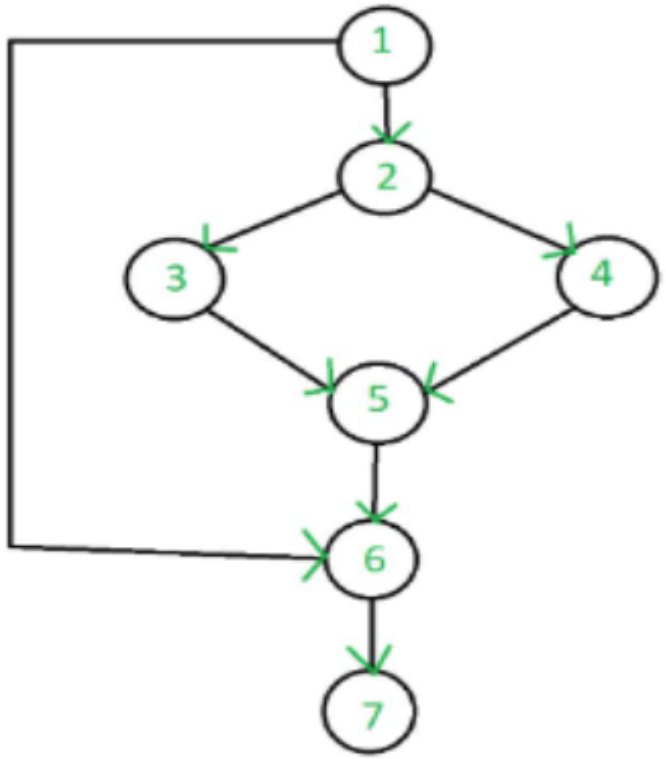
Basis Path Testing

- Basis Path Testing is a white-box testing technique based on the control structure of a program or a module.
- Basis path testing is a technique of selecting the paths in the control flow graph, that provide a basis set of execution paths through the program or module.
- To design test cases using this technique, four steps are followed:
- 1. Construct the Control Flow Graph
- 2. Compute the Cyclomatic Complexity of the Graph
- 3. Identify the Independent Paths
- 4. Design Test cases from Independent Paths

Example 1:

```
if  A = 10 then
  if B > C
     A = B
  else A = C
  endif
  endif
print A, B, C
```





Control Flow Graph

Cyclomatic Complexity:

1. $V(G) = E-N+2P$

   Where, E→ No. of edges

   N→ No. of nodes

   P → No. of exit points

   $V(G) = 8-7+2(1)$

   $= 1+2$

   $= 3$

2. $V(G) = P+1$

   Where P→ Decision-making nodes

   $= 2+1$

   $= 3$

3. $V(G) = R+1$

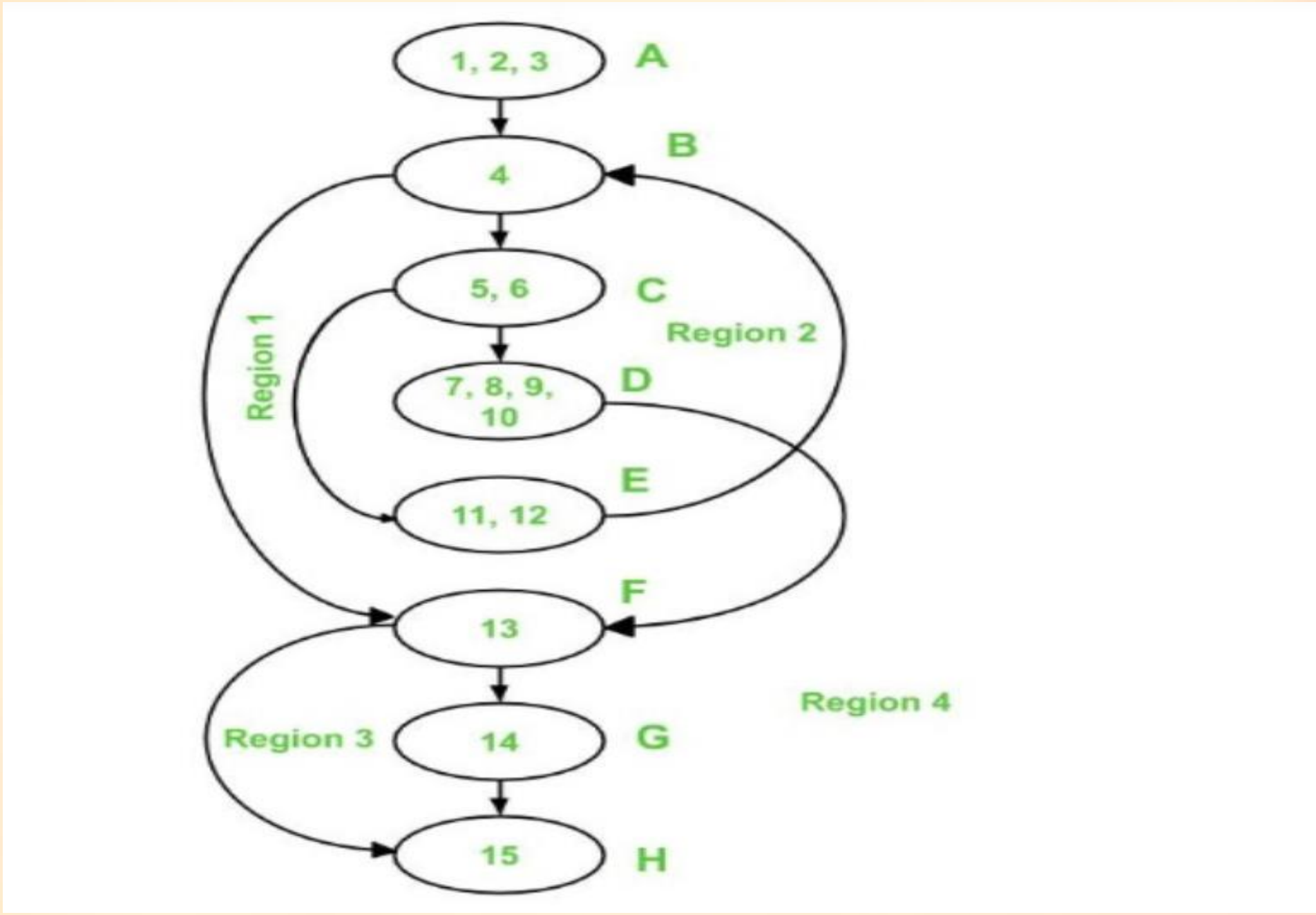
   Where, R→ No. of regions

   $= 2+1$

   $=3$

Example 2: Program to find whether a given number is prime or not.

```
int main()
{
    int n, index;
1    cout << "Enter a number: " <> n;
3    index = 2;
4    while (index <= n - 1)
5    {
6        if (n % index == 0)
7        {
8            cout << "It is not a prime number" << endl;
9            break;
10        }
11        index++;
12    }
13    if (index == n)
14        cout << "It is a prime number" << endl;
15  } // end main
```


1, 2, 3
A
4
B
5, 6
C
Region 2
Region 1
7, 8, 9, 10
D
11, 12
E
13
F
Region 4
Region 3
14
G
15
H

Cyclomatic Complexity:

1. V(G) = E-N+2P
   E=10, N=8, P=1
   V(G) = 10-8+2*1
   = 2+2
   = 4

2. V(G) = P+1
   = 3+1 (Node 1,6,13)
   = 4

3. V(G) = R+1
   = 3+1
   =4

Test Coverage Metrics

- Test coverage metrics are **measurements used in software testing** to check how much of the code, requirements, or functionality has been tested.
- They help ensure that **no important part of the system is left untested**.

Types of Test Coverage Metrics

- **a) Code Coverage**
- Measures how much of the program's source code is executed during testing.
- **Statement Coverage** → % of lines/statements executed
- **Branch Coverage** → % of decision branches (if-else, switch) executed
- **Condition Coverage** → % of boolean expressions tested as both TRUE and FALSE
- **Example:**
  If a program has 100 lines and tests execute 80 lines → **80% statement coverage**.

- **b) Requirement Coverage**
- Measures how many of the **requirements or features** have been tested.
- Ensures each requirement has at least one test case.
- **Example:**
  If there are 20 requirements and only 15 are tested → **75% requirement coverage**.
- **c) Path Coverage**
- Measures whether **all possible execution paths** through the program are tested.
- Stronger than branch coverage, ensures complex flows are tested.

- **d) Function Coverage**
- Checks whether **all functions/methods** in the code are tested at least once.

- **e) Test Case Coverage**
- Measures how many of the **planned test cases** were actually executed.

What is Statement Coverage?

- It is one type of white box testing technique that ensures that all the statements of the source code are executed at least once.
- It covers all the paths, lines, and statements of a source code.
- It is used to design test box cases where it will find out the total number of executed statements out of the total statements present in the code.
- FORMULA:
- *Statement coverage = (Number of executed statements / Total number of statements in source code) * 100*

- *EXAMPLE:1*

- *Read A*
  Read B
  if A > B
  Print “A is greater than B”
  else
  Print "B is greater than A"
  endif

Step 1: Count Executable Statements

1. `Read A`
2. `Read B`
3. `Print “A is greater than B”` (executed if condition TRUE)
4. `Print "B is greater than A"` (executed if condition FALSE)

👉 **Total = 4 executable statements**

Step 2: Test Cases and Coverage

- **Test Case 1:** A = 10, B = 5
  - Executes: `Read A`, `Read B`, `Print “A is greater than B”`
  - Covered = 3/4 statements
  - **Coverage = (3/4) × 100 = 75%**
- **Test Case 2:** A = 5, B = 10
  - Executes: `Read A`, `Read B`, `Print "B is greater than A"`
  - Covered = 3/4 statements
  - **Coverage = 75%**
- **Both Test Cases Together:**
  - All 4 statements get executed across the test set
  - **Coverage = (4/4) × 100 = 100%**

✅ Final Answer

- With **only one test case → 75% statement coverage**
- With **both true & false test cases → 100% statement coverage**

- *EXAMPLE-2*

- ```
  print (int a, int b)
  {
      int sum = a + b;
      if (sum > 0)
         print ("Result is positive")
     else
        print ("Result is negative")
  }
  ```
  ```

What is Branch Coverage?

- Branch coverage in unit testing is a metric that measures the percentage of branches (decision points) in the source code that have been executed during the testing process.
- It indicates how well the test cases navigate through different possible outcomes of conditional statements, helping evaluate the thoroughness of testing.
- FORMULA:

- *Branch Coverage = ((Number of Executed Branches) ÷ (Total Number of Branches)) ×100%*

- EXAMPLE-1
- checkAge(int age) {
- if (age >= 18)
- print("Eligible to Vote");
- else
- print("Not Eligible to Vote");

- print("Print the Age: %d" + age);
- }
- **Step 1: Identify Branches**
- The program has **1 decision**: if(age>=18)

- **Step 3: Test Cases for Branch Coverage**
- **Test Case 1:** age = 20 → Takes **True branch**
- **Test Case 2:** age = 15 → Takes **False branch**

Step 4: Branch Coverage Formula

$$\text{Branch Coverage (\%)} = \frac{\text{Number of Executed Branches}}{\text{Total Number of Branches}} \times 100$$

- Total branches = 2
- If we run **only one test case (age=20)** → Executed = 1 → Coverage = $\frac{1}{2} \times 100 = 50\%$
- If we run **both test cases (age=20, age=15)** → Executed = 2 → Coverage = $\frac{2}{2} \times 100 = 100\%$

✅ **Final Answer:**

- With **one test case → 50% branch coverage**
- With **two test cases (≥18 and <18) → 100% branch coverage**

Condition Coverage Testing

- In the software condition coverage testing every Boolean expression described in the conditions expression is evaluated to both true and false outcomes.
- As a result, it ensures that both the branches in a decision statement are tested.
- In case, a decision statement comprises various conditions namely OR, and AND, the condition coverage testing confirms that all the various combinations of the conditions are included in the test cases.
- Formula
- Condition Coverage = (Total count of conditions executed / Total count of conditions in the source code) * 100

- EXAMPLE-1

```
if (x > 0 && y < 10)
    printf("Valid");
else
    printf("Invalid");
```

- Conditions:
  1. `(x > 0)`
  2. `(y < 10)`

👉 For **Condition Coverage**, each of these must be **TRUE once** and **FALSE once**, across the test set.

Possible Test Cases:

- Case 1: x = 5, y = 5 → `(x > 0)` = TRUE, `(y < 10)` = TRUE → overall TRUE
- Case 2: x = -2, y = 5 → `(x > 0)` = FALSE, `(y < 10)` = TRUE → overall FALSE
- Case 3: x = 5, y = 15 → `(x > 0)` = TRUE, `(y < 10)` = FALSE → overall FALSE

👉 Now:

- `(x > 0)` → TRUE in Case 1 & 3, FALSE in Case 2
- `(y < 10)` → TRUE in Case 1 & 2, FALSE in Case 3

✅ Each condition was TRUE at least once and FALSE at least once → **100% Condition Coverage**

⚠️ Notice: You don't need to test **all combinations** (like x=-2, y=15) for condition coverage (that would be **multiple condition coverage**). You just need **each condition to flip both ways**.

Requirement Coverage

- **Definition:**
  Requirement coverage measures **how many of the documented requirements (from SRS or user stories)** have been tested by at least one test case.
- **Goal:**
  Ensure that **every requirement** has been validated, so nothing the customer asked for is missed.
- **Formula:**
- Requirement Coverage (%)=Total no. of requirementsNo. of requirements tested×100

- **Simple Example**
- Suppose we have a **Login System** with 5 requirements:
- User must enter a valid username.
- User must enter a valid password.
- System must lock account after 3 failed attempts.
- System must allow password reset through email.
- Login must redirect to the dashboard after success.
- **Test Cases Written:**
- TC1 → Tests username validation (Req 1)
- TC2 → Tests password validation (Req 2)
- TC3 → Tests account lock after 3 failures (Req 3)
- TC4 → Tests successful login redirects to dashboard (Req 5)
- 👉 **Requirement 4 (Password reset)** has **no test case** written.

- **Requirement Coverage:**
- Total Requirements = 5
- Tested Requirements = 4
- Coverage=4/5×100=80%

Path Coverage Testing Metrics

- Path Coverage Testing Metrics is a **white-box testing technique** used in software testing to ensure that **all possible paths** through the program (or a specific module/function) are tested at least once.
- It is a stronger coverage criterion than statement or branch coverage because it checks combinations of decisions, not just individual ones.
- **Path Coverage = (Number of paths executed / Total number of possible paths) × 100%**
- A *path* is a unique sequence of statements and decisions from the start of the program to the end.
- Path coverage ensures that all **independent paths** (based on control flow) are tested.

```
#include <stdio.h>
int main() {
    int x = 5, y = 10;
    if (x > 0) {
        if (y > 5)
            printf("Path 1\n");
        else
            printf("Path 2\n");
    }
    else {
        if (y > 0)
            printf("Path 3\n");
        else
            printf("Path 4\n");
    }
    return 0;
}
```

- **Paths in the Program:**
- **x > 0 → y > 5** → Path 1
- **x > 0 → y ≤ 5** → Path 2
- **x ≤ 0 → y > 0** → Path 3
- **x ≤ 0 → y ≤ 0** → Path 4
- ☞ Total possible paths = **4**

🔹 Path Coverage Metric:

If only 2 test cases are executed (say, Path 1 and Path 3):

$$\text{Path Coverage} = \frac{2}{4} \times 100\% = 50\%$$

If all 4 test cases are executed:

$$\text{Path Coverage} = \frac{4}{4} \times 100\% = 100\%$$

🔹 Test Cases for Path Coverage:

- Case 1: `x = 5, y = 10` → Path 1
- Case 2: `x = 5, y = 3` → Path 2
- Case 3: `x = -1, y = 2` → Path 3
- Case 4: `x = -2, y = -1` → Path 4

Path Notation

- is a way of writing **program paths** using the **Control Flow Graph (CFG)** of the program.
- **Control Flow Graph (CFG)**
- In white-box testing, the program is often drawn as a **graph**.
- **Nodes** = statements or blocks of code.
- **Edges** = control flow (the direction of execution).
- Each edge/path is labeled, often with letters (A, B, C…).

EXAMPLE-1

```
if (x > 0)       // A
  y = 1;         // B
else
  y = -1;        // C

print(y);        // F
```

CFG could look like:

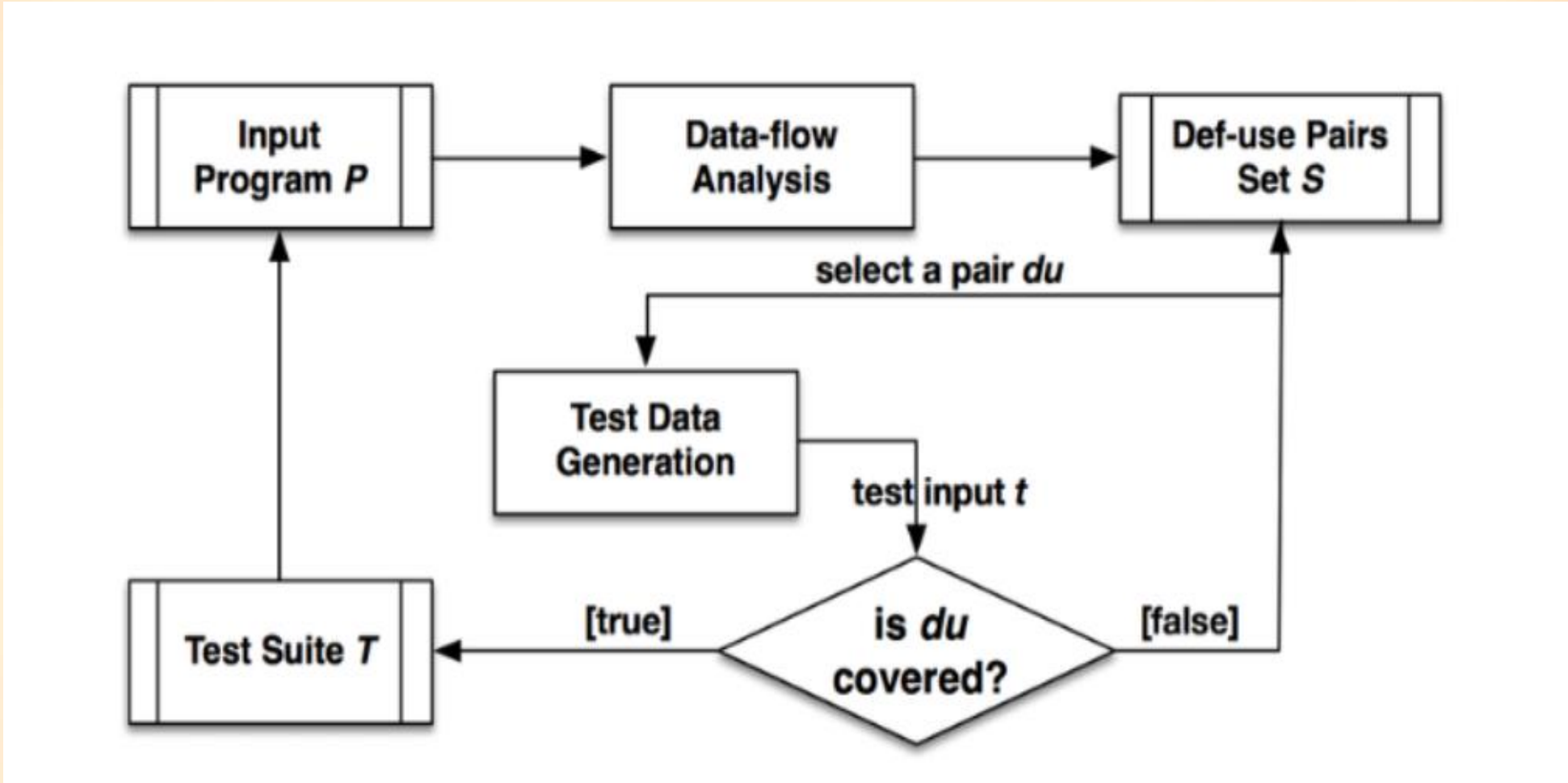
- Start → A (decision)
- True branch → B → F
- False branch → C → F

So paths are:

- **1A – 2B – 4F** (if condition true)
- **1A – 3C – 4F** (if condition false)

Data Flow testing

- Data flow testing is a type of structural testing used to analyze the flow of data in the program.
- It is a method that is used to find the test paths of a program according to the locations of definitions and uses of variables in the program.
- It is concerned with:
- ● Statements where variables receive values,
- ● Statements where these values are used or referenced.
- Data flow testing can be done using one of the following two techniques:
- ● Control flow graph
- ● Making associations between data definition and usage


Input
Program P
Data-flow
Analysis
Def-use Pairs
Set S
select a pair du
Test Data
Generation
test input t
is du
covered?
[true]
[false]
Test Suite T

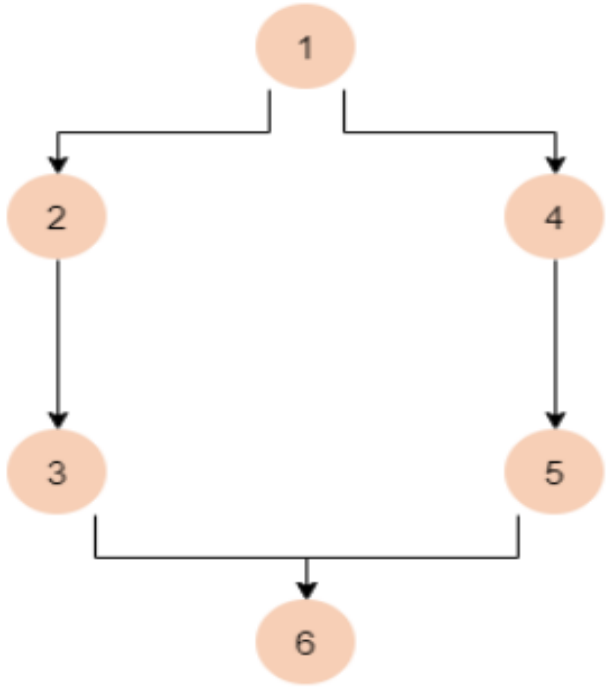
- **Input Program P**
- This is the program or code that you want to test.
- It is the subject under test, which will undergo **data-flow analysis** to detect where variables are defined (assigned a value) and used.
- **Data-flow Analysis**
- Analyzes the program to determine **Def-Use (DU) pairs** of variables.
- **Definition (Def):** when a variable is assigned a value.
- **Use:** when that variable's value is later read/used.
- The analysis identifies potential data dependencies that need to be tested.

- **Def-use Pairs Set S**
- The result of the data-flow analysis is a **set of all definition–use pairs (S)**.
- Each pair shows where a variable is defined and later used in the program.
- Example:
- int x = 5; // definition of x
- y = x + 2; // use of x
- **Select a Pair du**
- From the set S, the process selects one definition–use pair (du) at a time for coverage checking.
- The goal is to ensure that for each pair, a test input exists which makes the execution path from the definition to the use happen.

- **Test Data Generation**
- For the selected pair(du) appropriate **test input t** is generated.
- The input is chosen to force the execution of the code path that covers the definition and use of the variable.
- **Is du Covered?**
- This decision block checks if the generated test input actually causes the **definition to reach the use** without being redefined in between.
- If **True** → the DU pair is covered.
- If **False** → go back and try another test input or pair

Control flow graph example

```
1. input(x)
2. if(x>5)
3.       z = x + 10
4. else
5.       z = x - 5
6. print("Value of Z: ", z)
```



Variable Name	Defined At	Used At
x	1	2
z	3, 5	6

- **Making associations**
- In this technique, we make associations between two kinds of statements:
- ● Where variables are defined
- ● Where those variables are used

```
1. input(x)
2. if(x>5)
3.      z = x + 10
4. else
5.      z = x - 5
6. print("Value of Z: ", z)
```

For the above snippet of pseudo-code, we will make the following associations:

- (1, (2,t), x): for the true case of the IF statement in line 2
- (1, (2,f), x): for the false case of IF statement in line 2
- (1, 3, x): variable x is being used in line 3 to define the value of z
- (1, 5, x): variable x is being used in line 5 to define the value of z
- (3, 6, z): variable z is being used in line 6, which is defined in line 3
- (5, 6, z): variable z is being used in line 6, which is defined in line 5

The first two associations are for the IF statement on line 2. One association is made if the condition is true, and the other is for the false case.

Now, there are two types of uses of a variable:

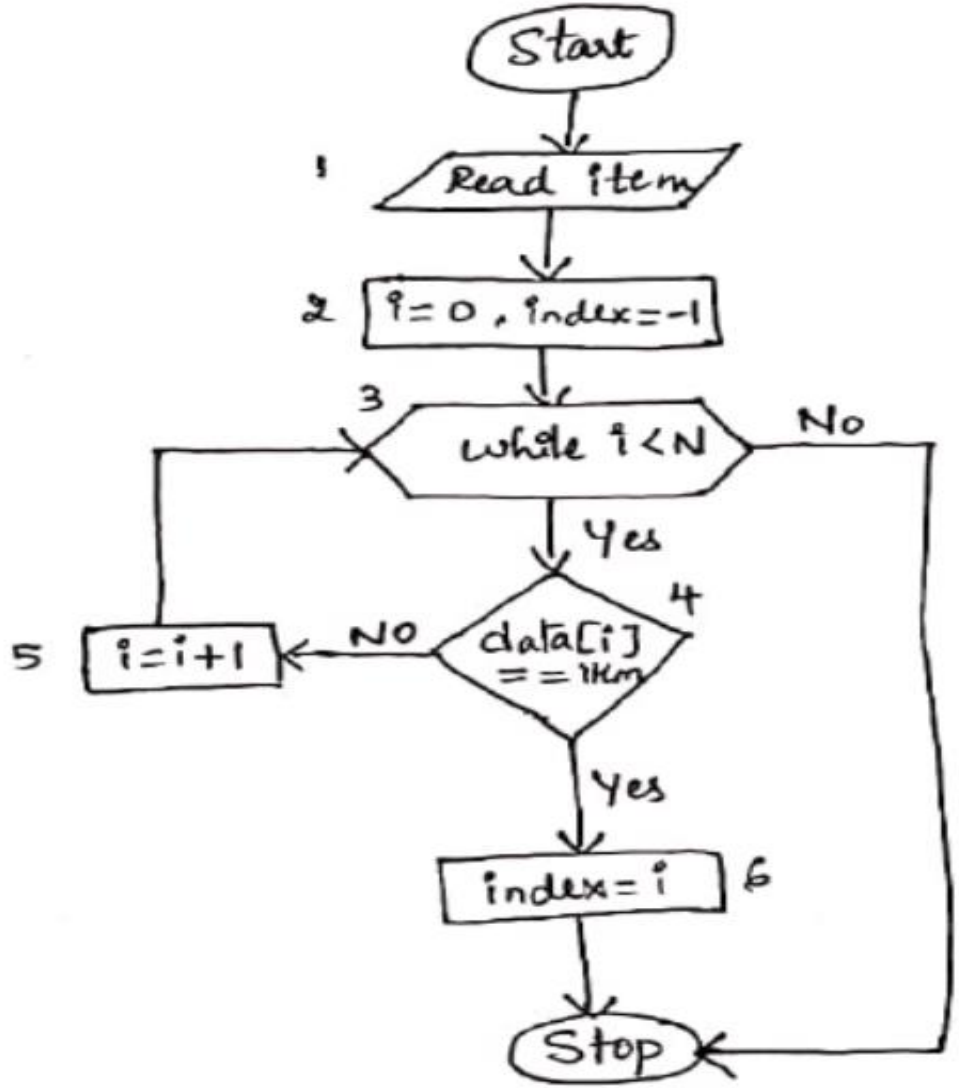
- **predicate use**: the use of a variable is called p-use. Its value is used to decide the flow of the program, e.g., line 2.
- **computational use**: the use of a variable is called c-use when its value is used compute another variable or the output, e.g., line 3.

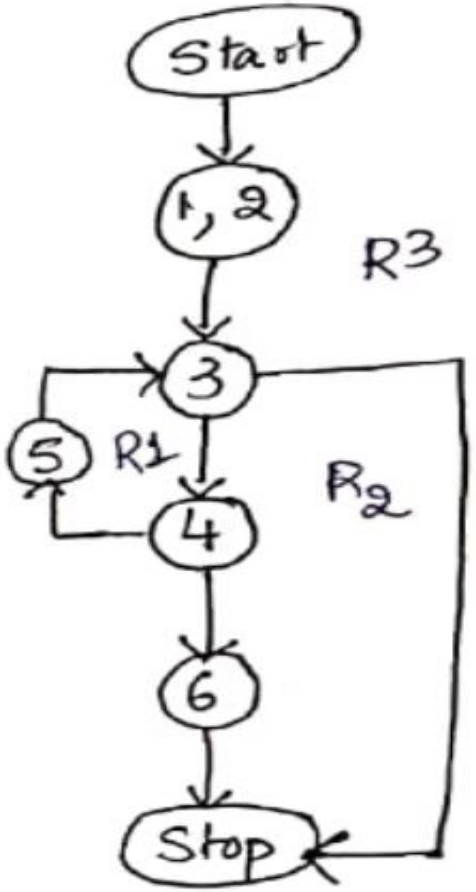
- After the associations are made, these associations can be divided into these groups:
- ● All definitions coverage
- ● All p-use coverage
- ● All c-use coverage
- ● All p-use, some c-use coverage
- ● All c-use, some p-use coverage
- ● All use coverage
- Once the associations are divided into these groups, the tester makes test cases and examines each point.
- The statements and variables found to be extra are removed from the code.

Data Flow Testing Coverage:

- The coverage of data flow in terms of "sub-paths" and "complete paths" may be categorized under the following types:
- **All definition coverage:** Covers "sub-paths" from each definition to some of their respective use.
- **All definition-C use coverage:** "sub-paths" from each definition to all their respective C use.
- **All definition-P use coverage:** "sub-paths" from each definition to all their respective P use.
- **All use coverage:** Coverage of "sub-paths" from each definition to every respective use irrespective of types.
- **All definition use coverage:** Coverage of "simple sub-paths" from each definition to every respective use.

Flowchart for Linear Search Algorithm.





Control graph

Variables	Defined	used.
item	1	4
i	2	3, 4, 6
index	2, 6	—
N	—	3

du-pairs

(1, 4)

(2, 3) (2, 4)

(2, 6)

du-paths

{ 1, 2, 3, 4 }

{ 2, 3 }

{ 2, 3, 4 } { 2, 3, 4, 6 }

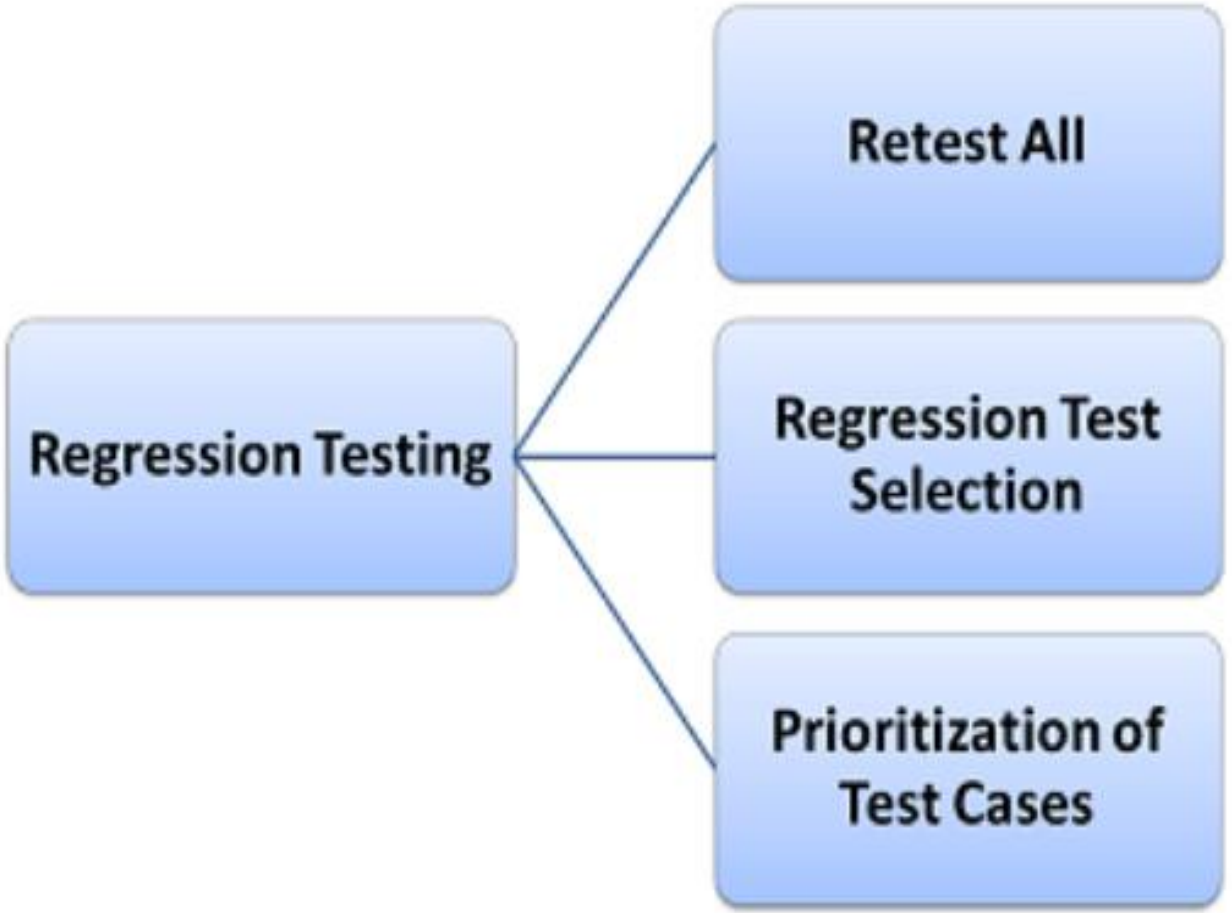
REGRESSION TESTING

- Regression testing is a software testing practice that ensures an application still functions as expected after any code changes, updates, or improvements.
- **corrective regression testing**: It refers to regression testing of a program obtained by making corrections to the previous versions.
- **progressive regression testing:** It refers to regression testing of a program obtained by adding new features.

- Typically, regression testing is applied under these circumstances:

- A new requirement is added to an existing feature
- A new feature or functionality is added
- The codebase is fixed to solve defects
- The source code is optimized to improve performance
- Patch fixes are added
- Changes in configuration

Regression testing can be performed using the following techniques:



- **1. Re-test All:**
- Re-Test is one of the approaches to do regression testing.
- In this approach, all the test case suits should be re-executed.
- Here we can define re-test as when a test fails, and we determine the cause of the failure is a software fault.
- If the fault is reported, we can expect a new version of the software in which the defect is fixed.
- In this case, we will need to execute the test again to confirm that the fault is fixed.
- This is known as re-testing.
- Some will refer to this as confirmation testing.
- The re-test is very expensive, as it requires enormous time and resources.

- **2. Regression test Selection:**
- In this technique, a selected test-case suit will execute rather than an entire test-case suit.
- The selected test case suits are divided into two cases

● Reusable test cases can use in succeeding regression cycles.

● Obsolete test cases can't use in succeeding regression cycles.

3. Prioritization of test cases:

Prioritize the test case depending on business impact, and critical and frequent functionality used.

The selection of test cases will reduce the regression test suite.

Need for Regression Testing:

- Regression testing is also needed when a subsystem is modified to generate a new version of an application.
- •When one or more components of an application are modified the entire application must also be subject to regression testing.
- •In some cases regression testing might be needed when the underlying hardware changes.
- In this case, regression testing is performed despite any change in the software.
- Regression testing can be applied in each phase of software development.

REGRESSION TEST PROCESS

- **Test revalidation:** refers to the task of checking which tests for P remain valid for P'.
- Revalidation is necessary to ensure that only tests that are applicable to P' are used during regression testing.
- **Test Selection:** The identification of tests that traverse modified portions of P' is often referred to as test selection and sometimes as the regression-test selection (RTS) problem.
- There are 2 techniques for test selection.
- 1. **Test minimization2. Test prioritization**
- Test minimization discards tests as seemingly redundant with respect to some criteria.
- Test prioritization refers to the task of prioritizing tests based on some criteria.
- Test selection can be achieved by selecting a few tests from a prioritized list.

- **The test setup:** refers to the process by which the application under test is placed in its intended, or simulated, environment ready to receive data and able to transfer any desired output information.
- Test setup requires simulators as a replacement for real devices and software and hardware environmental setup.
- Eg1: a heart simulator is used while testing a commonly used heart control device known as the pacemaker
- Eg2: The test setup process and the setup for automobile engine control software are quite different from that of a cellphone.
- **Test sequencing:** often becomes important for an application with an internal state that is continuously running.
- Eg: Banking software, web service, and engine controller are examples of such applications.

- **Test execution:** Once the testing infrastructure has been set up, tests selected, revalidated, and sequenced, it is time to execute them.
- •This task is often automated using a generic or a special-purpose tool.
- •General purpose tools are available to run regression tests for applications such as web service
- •Embedded systems, due to their unique hardware requirements, often require special-purpose tools that input a test suite and automatically run the application

- **Output Comparison:**

•Each test needs verification.

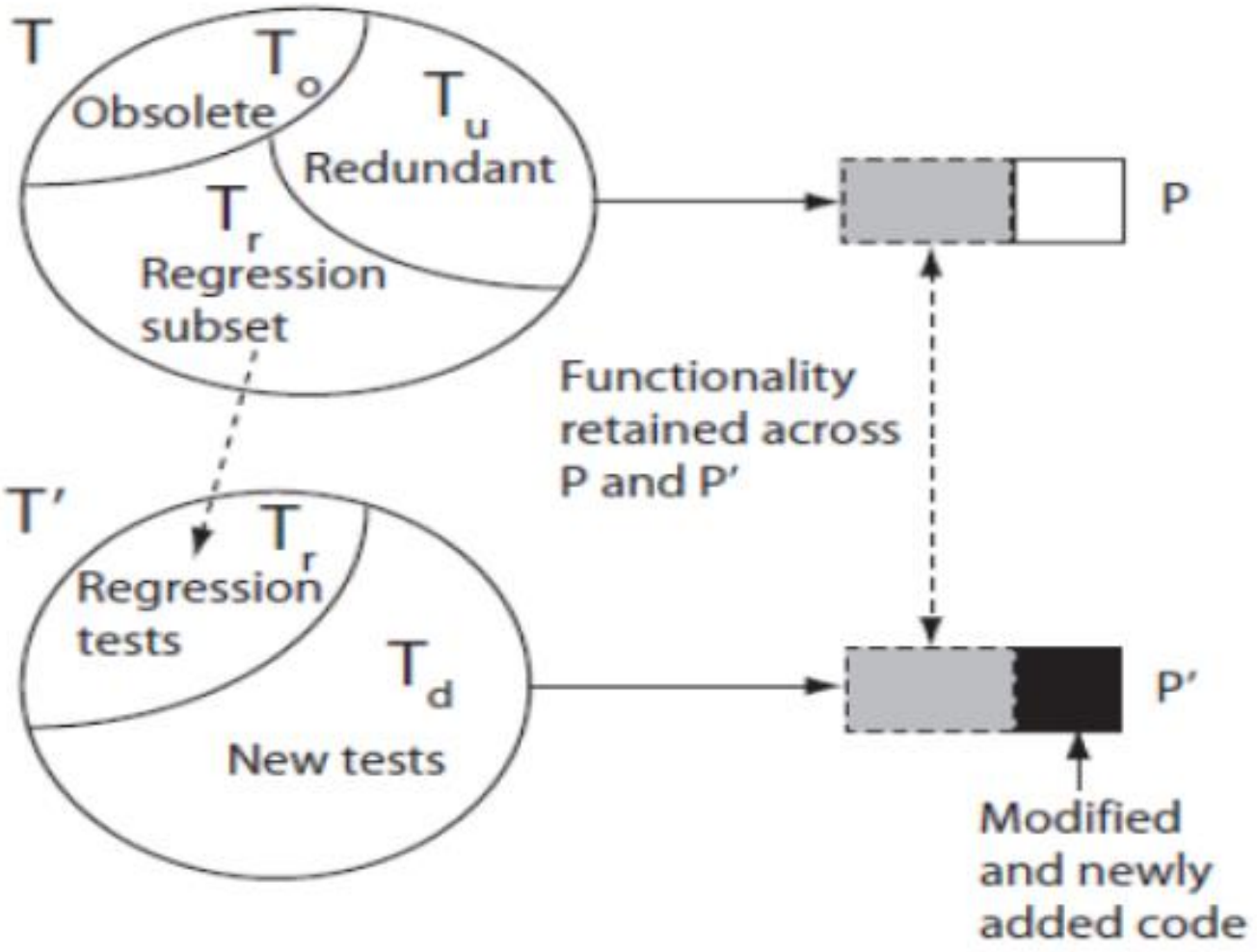
This is done automatically using the test execution tool

•However, this might not be a simple process, especially in embedded systems.

•Sometimes for embedded systems we can’t use generic tools

• goal for test execution is to measure an application’s performance.

•Eg: one might want to know how many requests per second can be processed by a web service Dept. of ISE, NHCE18


T
T o
Obsolete
T u
Redundant
T r
Regression
subset
T'
T r
Regression
tests
T d
New tests
Functionality
retained across
P and P'
P
P'
Modified
and newly
added code

TEST SELECTION FOR REGRESSION TESTING

- 1**. Test All:** We use T'=T-To to test all strategies
- Disadvantage: Time consumption
- **2. Random Selection:** Tests are selected randomly from the set T-To.
- The tester decides how many tests to select depending on the level of confidence required and the available time.
- Disadvantage:Some of the sampled tests might bear no relationship to the modified code

- **3. Selecting Modification-Traversing Tests**
- We use methods to determine the desired subset and aim at obtaining a minimal regression test suite.
- Techniques that obtain a minimal regression-test suite without discarding any test that will traverse a modified statement are called safe RTS techniques
- Advantage: In a time crunch, testers execute only smaller no of tests
- Disadvantage: Sophisticated techniques to select modification-traversing tests require automation.

- **4. Test Minimization**
- It reduces the size of the regression test suite but discards the redundant tests
- Disadvantage: Tests discarded by the minimization algorithm must be reviewed carefully before discarding

5. Test Prioritization

A suitable metric is used to rank all the tests. A test with the highest rank has the highest priority

Eg: let R1, R2, and R3 require carried unchanged from P to P|. Rank-R2-R3-R1 and Td are new tests that need to be tested
{t1,t2,t3,t4,t5}-subset
t1-R1,
t2&t3-R2,
t4&t5-R3
We can prioritize as t2,t3,t4,t5,t1

6. Test selection using Execution Trace

Step 1: Given P and test set T, find the execution trace of P for each test in T.

Step 2: Extract test vectors from the execution traces for each node in the CFG of P

Step 3: Construct syntax trees for each node in the CFGs of P and P'. This step can be executed while constructing the CFGs of P and P's.

Step 4: Traverse the CFGs and determine the subset of T appropriate for regression testing of P'.

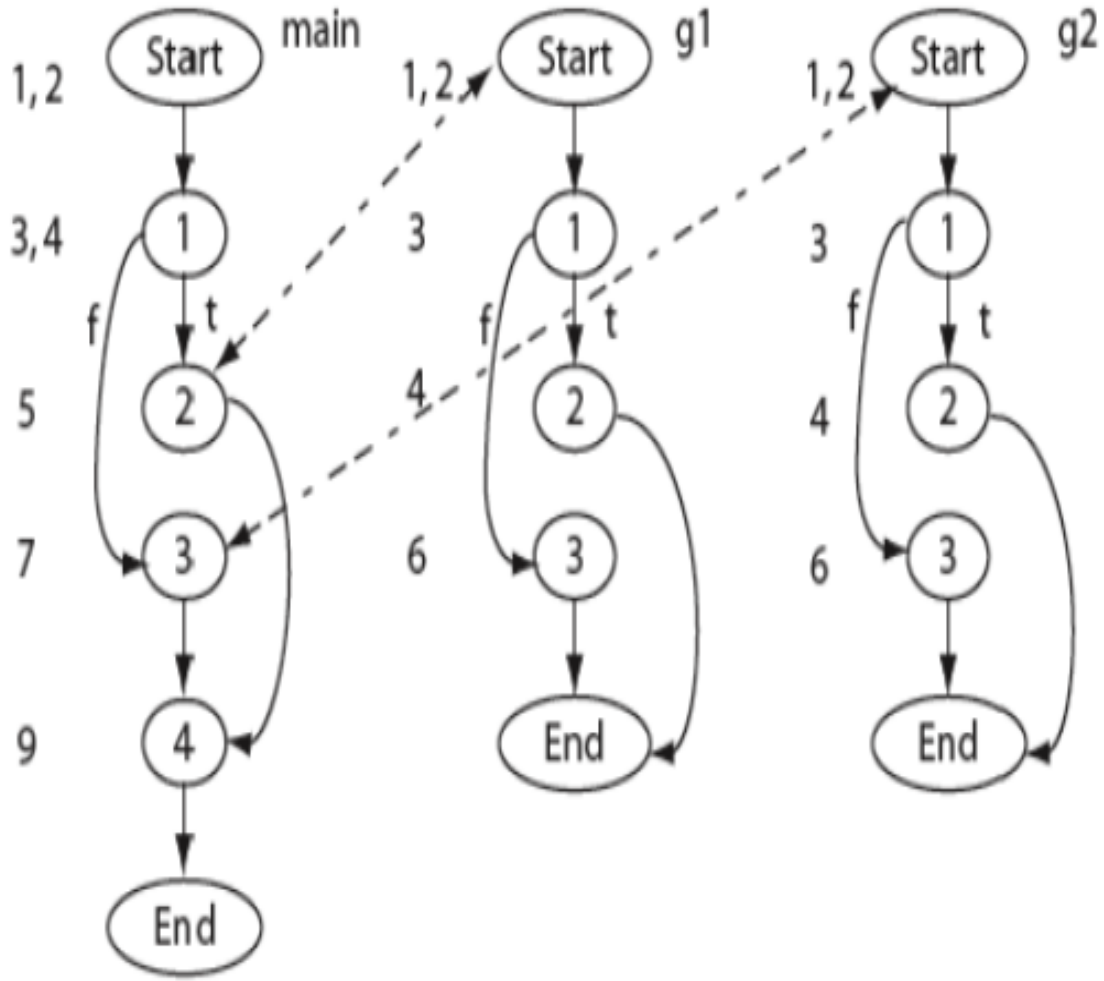
- Let G=(N, E) denote the CFG of program P
- N is a finite set of nodes
- E a finite set of edges connecting the nodes.
- Let Tno be the set of all valid tests for P'

```
1  main(){
2  int x,y,p;
3  input (x,y);
4  if (x<y)
5    p=g1(x,y);
6  else
7    p=g2(x,y);
8  endif
9  output (p);
10 end
11 }
```

```
1  int g1(int a, b){
2  int a,b;
3  if(a+ 1==b)
4    return(a*a);
5  else
6    return(b*b);
7  }
```

```
1  int g2 (int a, b){
2  int a,b;
3  if(a==(b+1))
4    return(b*b);
5  else
6    return(a*a);
7  }
```

Example program



CFG for example program

Test (t)	Execution trace ($trace(t)$)
t_1	main.Start, main.1, main.2, g1.Start, g1.1, g1.3, g1.End, main.2, main.4, main.End.
t_2	main.Start, main.1, main.3, g2.Start, g2.1, g2.2, g2.End, main.3, main.4, main.End.
t_3	main.Start, main.1, main.2, g1.Start, g1.1, g1.2, g1.End, main.2, main.4, main.End.

7. Test Vector and Syntax Trees

A test vector for node n, denoted by test(n), is the set of tests that traverse node n in the CFG

	Test vector (*test*(*n*)) for node *n*			
Function	**1**	**2**	**3**	**4**
main	t_1, t_2, t_3	t_1, t_3	t_2	$t_1, t_2\ t_3$
g1	t_1, t_3	t_3	t_1	–
g2	t_2	t_2	None	–

Syntax trees:

- A syntax tree is constructed for each node of CFG(P) and CFG(P').